Reducing the Cost of Grammar-based Testing using Pattern Coverage

Cleverton Hentz[1], Jurgen J. Vinju[2,3,4], and Anamaria M. Moreira[5]

[1] Federal University of Rio Grande do Norte, Brazil
[2] Centrum Wiskunde & Informatica, The Netherlands
[3] Eindhoven University of Technology, The Netherlands
[4] INRIA Lille Nord Europe, France
[5] Federal University of Rio de Janeiro, Brazil
chentz@ppgsc.ufrn.br, jurgen.vinju@cwi.nl, anamaria@dcc.ufrj.br

Abstract. In grammar-based testing, context-free grammars may be used to generate relevant test inputs for language processors, or meta programs, such as programming language compilers, refactoring tools, and implementations of software quality metrics. This technique can be used to test these meta programs, but the amount of sentences, and syntax trees thereof, which needs to be generated to obtain reasonable coverage of the input language is exponential.
Pattern matching is a programming language feature used often when writing meta programs. Pattern matching helps because it automates the frequently occurring task of detecting shapes in, and extracting information from syntax trees. However, meta programs which contain many patterns are difficult to test using only randomly generated sentences from grammar rules. The reason is that statistically it is uncommon to directly generate sentences which accidentally match the patterns in the code.
To solve this problem, in this paper we extract information from the patterns in the code of meta programs to guide the sentence generation process. We introduce a new coverage criterion, called Pattern Coverage, which focuses on providing a test strategy to reduce the amount of test necessary cases, while covering the relevant parts of the meta program. An initial experimental evaluation is presented and the result is compared with traditional grammar-based testing.

Keywords: Software test, meta program, pattern matching, grammar-based testing.

1 Introduction

Meta programs are tools which read sentences of software languages, such as programming languages, and produce any kind of output [9]. Examples of meta programs are compilers, interpreters, refactoring tools, static analysis tools, and source code metrics tools. The amount and diversity of such tools are growing as processing power and large memory become available to the machines on which

software is being developed, but verifying such tools is quite a challenge. Even simple metric tools are known to contain many bugs [24].

Meta programs for real programming languages are complex and hard to prove or test because the languages are big (more than 400 context-free grammar rules is quite normal) and their semantics is often unclear. Apart from some notable exceptions [23], proofs of correctness of meta programs are not to be expected. Therefore we wish to quickly find common errors in meta programs by exercising their code based on test generation.

Grammar-based testing is a preferred approach for testing meta programs [5, 20]. The input syntax of most meta programs can be modeled precisely using context-free grammars (CFG). From such CFG specifications we can define sets of input sentences which satisfy different coverage criteria of the input language. However, the amount of sentences necessary to cover an entire language is intractable. For Java, for example, if we consider only sentences generated by derivation tree up to height 7 the amount of sentences that can be generated is around 46.26×10^9. If we increase this value to 10, the amount goes to around 9.43×10^{43}. But, even if we can run a Java processor on such corpus of inputs in reasonable time, chances are the corpus will still not lead to good coverage of the code of the processor itself as some constructs will only appear deeper in the grammar.

The reason for the bad coverage is that a context-free grammar describes all possible inputs for a meta program, but it does not specify precisely which part of the language is used by the given meta program or how different parts of the language are distributed over the meta program. A random distribution over an input grammar will therefore typically not generate an effective test set for a given meta program. The grammars of real programming languages have hundreds of (recursive) rules, generating a super-exponential amount of syntactically correct inputs. Due to the size of such grammars, statistically it is hard to generate exactly the right combination to cover a meta program with a limited (feasible) set of test cases. To further aggravate this problem, some important semantic information, which is dealt with in the meta program, is often missing from context-free grammar descriptions.

Pattern matching is an interesting feature which is often used during the development of meta programs. With it, it is simpler to describe specific situations over the input language, simplifying the implementation of this kind of programs. Many meta programming languages (specialized languages for the development of meta programs) and libraries are based on expressive forms of pattern matching (Haskell [15], Scala [28], TOM [3], ASF+SDF [6], Maude [8], StrategoXT [7], ELAN [4] and Rascal [18, 19]).

The challenge we address is: given a grammar which describes syntactically correct inputs, to generate a minimal amount of inputs which will cover the meta program. Our contribution is the notion of pattern coverage which links grammar coverage to conditional coverage in meta programs which use pattern matching. We propose the use of patterns as reference for the definition of test data and a pattern-based coverage criterion which defines a set of test data requirements

from a set of language patterns. These patterns may be extracted from different sources: from language or meta program specifications, from the patterns used in the code of the meta program, or even from a non pattern-based meta program from which some mining of patterns has been carried out.

The main contributions of this paper are then:

- the definition of the *Pattern Coverage* criterion;
- an initial algorithm for generating sentences for *Pattern Coverage*;
- an empirical evaluation of the relation between *Pattern Coverage* and existing notions of grammar coverage.

We conduced the evaluation using two meta programs that implement the Cyclomatic Complexity algorithm [25] for Java and Pico [12] languages. To evaluate and compare the pattern coverage criterion we used the mutation technique [11] to evaluate the generated test sets. As result of pattern coverage use we could observe a reduction on the test set size and the preservation of the quality level for the pattern coverage test set.

The remainder of this article is organized as follows. Section 2 presents the basic theoretical foundation used to the work presentation. Section 3 define the pattern coverage criterion and present the test data generation algorithm. The Section 4 present the evaluation process and the results obtained during the process. Related work is discussed in Section 5, followed by concluding remarks in Section 6.

2 Background

In this section we briefly introduce and discuss the theoretical background on which this paper is based. First, in Section 2.1, some grammar-based testing concepts are introduced and discussed. Then, we define pattern matching, the mechanism used to simplify the structure of meta programs and that our proposal uses to make meta program testing more efficient.

2.1 Grammar-based Testing

There are many different criteria for software testing in the literature. They may be classified in different (often orthogonal) ways, depending on: the stage of the software development in which they are to be applied (e.g., unit, system or regression testing), the reference artifact used for test design (e.g., white and black box testing), the kind of abstraction used to extract requirements from the reference artifact (e.g., graphs, logical expressions or grammars).

As pointed out in [1], the most important classification when it comes to define or chose a criterion is the used abstraction. As our focus here are meta programs, and the classical way of dealing with criteria-based test design for meta programs is through *grammar-based coverage criteria*, where tests are derived from grammar descriptions of the software or of some software artifact. As these criteria are defined in terms of grammar components, in the following, we

present the definition of the most commonly used type of grammar (*Context-Free Grammar (CFG)*) where those components are introduced.

Definition 1 (Context-Free Grammar). *A Context-Free Grammar G is a 4-tuple (N, S, T, P) such that:*

1. *N is a finite set of* nonterminals*;*
2. *T is a finite set of* terminals*;*
3. *P is a finite subset of $N \times (T \cup N)^*$ called* production rules
4. *$S \in N$ is a start symbol.*

Every grammar G defines a language (set of sentences), which is noted $L(G)$. The sentences of the language are the sequences of terminals of G which can be derived from the initial symbol S by sequences of production rule applications (*derivations*)[6].

The grammar-based test design process starts with the application of the chosen criterion to the grammar to define a set of *test requirements*. These test requirements specify properties that need to be satisfied by the set of test cases (*test set*), and a test requirement is said to be *satisfied* if at least one of the test cases of the test set satisfies the specified property.

The typical grammar-based test criteria are defined in terms of terminals, production rules or language sentences or derivations.

Definition 2 (Grammar-based Coverage Criteria). *For a grammar $G = (N, S, T, P)$,*

- Terminal Coverage (TSC)*: for each terminal symbol $t \in T$ there exists exactly one test requirement: t occurs in the sentence, or, simply, t. So, $TSC = \{t \in T\}$ is the set of test requirements to cover this criterion;*
- Production Coverage (PDC)*: for each production rule $\rho \in P$ there exists exactly one test requirement: ρ is used in the derivation of the sentence, or, simply, ρ. So, $PDC = \{\rho \in P\}$ is the set of test requirements to cover this criterion;*
- Derivation Coverage (DC)*: for a grammar G, each derivation represented by $s \in L(G)$ one test requirement. So, the set $DC = \{s \in L(G)\}$ represents all test requirements to cover this criterion.*

While Terminal Coverage and Production Coverage usually require a small amount of test cases (test sentences) to be attained, Derivation Coverage is obviously impossible to achieve in most cases. A variant is often used [21, 27], then, which is to limit the length of the considered strings or derivations to some fixed limit and define as requirements the subset of $L(G)$ up to length n.

There are some tools that use those coverage criteria to generate test cases, for example the YouGen [14], XTextGen [13] and LGen [27] tools. These tools

[6] In the absence of grammar ambiguity, there is a one-to-one correspondence between sentences and derivation. We assume here non ambiguous grammars to simplify presentation.

N.	Instruction	Description
1	`int x := 3;`	Typed pattern, the `x` variable is bound to value `3`.
2	`if (add(l, r) := add(var("x"), var("y"))) println(l + "," r);`	Constructor pattern, the identifier `add` is the constructor and the variables `l` and `r` are bound to values `var("x")` and `var("b")`, respectively.
3	`for (/str x := add(var("x"), var("y"))) println(x);`	Deep matching pattern, the matching can happen on any level of the term.

Table 1. Example of some pattern-based instructions in Rascal.

basically receive a context-free grammar as input and some additional restrictions and generate a set of test cases (language sentences). These restrictions are necessary to limit the size of the resulting set, since, in general, the language associated to the input grammar is infinite.

2.2 Pattern Matching Mechanism

Pattern matching is a programming language feature. It is common to implement meta programs using pattern matching because it facilitates the description of interesting cases over the object language and dubs as a de-structuring variable binding mechanism.

Formally, the pattern matching problem is, given two terms s and l, determine if there is a substitution σ, such that $\sigma(l) = s$ [2]. The substitution itself, if it exists, represents the binding of variables which can be used by the program after the match.

Table 1 depicts a number of pattern types as used in the Rascal language [18]. Each line shows a value on the right and a pattern on the left of the match `:=` operator.

A most basic form of pattern, the congruence pattern, is an arbitrarily nested expression using only constructors and open variables. On table item 2, a pattern is used as a condition to the `if` statement. The congruence match succeeds because the value builds and `add` term, which the pattern matches literally. Then the children of the `add` term, `var("x")` and `var("y")` can be matched to the unconditional variables `l` and `r` respectively. The `l` and `r` variables may now be used in the program as normal variables, for example by printing them.

A more advanced pattern is the deep match on table item 3. It can be arbitrarily combined with any other match operator; in this case a typed variable. The current pattern will recursively traverse the matched value and succeed if a value of type `str` is found anywhere or fail otherwise. Such a pattern is *non-unitary* in the sense that it could match a single value in many ways (twice in this example). Using the `for` loop the programmer can iterate over all possible bindings of `x`, namely `"x"` and `"y"`.

Listing 1.1. Example of a pattern-based meta program written in Rascal [22].

```
1 int calcCC(Statement impl) {
2   int result = 1;
3   visit (impl) {
4     case \if(_,_) : result += 1;
5     case \if(_,_,_) : result += 1;
6     case \case(_) : result += 1;
7     case \do(_,_) : result += 1;
8     case \while(_,_) : result += 1;
9     case \for(_,_,_) : result += 1;
10    case \for(_,_,_,_) : result += 1;
11    case \foreach(_,_,_) : result += 1;
12    case \catch(_,_): result += 1;
13    case \conditional(_,_,_): result += 1;
14    case \infix(_,"&&",_) : result += 1;
15    case \infix(_,"||",_) : result += 1;
16  }
17  return result; }
```

3 Testing Meta Programs Using Pattern Coverage

Our proposal to deal with the original problem is to use the pattern information related to a meta program and with it generate a test set that will be used in the test process. This strategy may be applied on white-box testing, if the patterns are derived from the program, or black-box testing, if they are extracted from some more abstract model or specification of the meta program.

During this section, we will use a meta program (presented on Listing 1.1) as example to illustrate the concepts introduced on the section. This example is a typical implementation for the Cyclomatic Complexity (CC) [25] algorithm. The implementation of CC is coded in Rascal and was used by Landman [22] to calculate and analyze the correlation between the CC and source lines of code on a large corpus of Java code. For the purpose of the current paper this code is assumed to be correct. The `visit` statement generates a recursive traversal and each `case` represents a pattern to detect in the traversed tree, and after the colon (`:`) an action to perform when it matches (increasing a counter). Most of the cases define simple patterns which identify a particular node type, but two are more interesting, filtering only infix expressions with either `&&` or `||` as operators. The same code was used into our evaluation process and it will be detailed on section 4.

Our first step to address the test of meta programs is to define a coverage criterion that formalizes the requirements for a set of test cases. On Section 3.1 we introduce this criterion. After that, we introduce on Section 3.2 a prototype of algorithm to generate a test set that contains test cases to satisfy those requirements.

3.1 A Pattern-based Coverage Criterion

Our proposal is, given some pattern information corresponding to a meta program, to generate test cases exercising these patterns, i.e., to generate sentences that match them. To systematize this strategy, we define a new coverage criterion:

Definition 3 (Pattern Coverage). *For a pattern-based meta program m, and $\mathcal{P}_m$, the set with all instances of patterns of m, the test requirement set TR contains each element of the pattern set $\mathcal{P}_m$. Satisfaction of each of these requirements is attained by any subject which matches the corresponding pattern.*

To illustrate the pattern coverage definition, we can apply the definition to the CC implementation on Listing 1.1. For the given code the pattern set produced by collecting patterns is $\mathcal{P}_m = \{$`Statement impl`, `\if(_,_)`, `\if(_,_,_)`, ..., `infix(_,"||",_)` $\}$. Since in Rascal functions are dispatched by patterns, the formal parameter `Statement impl` is also considered a pattern, namely matching only abstract syntax trees of type `Statement`. The set $TR = \mathcal{P}_m$, and to cover TR the test data must include the set of java statements that match the corresponding patterns. For example, the pattern `\if(_,_)` would be covered by a program including the statement "`if (true) {}`", and an input which covers also the patterns for the Boolean operators would be "`if (true && false || true) { }`".

3.2 A Simple Algorithm to Generate Pattern Covering Test Sets

We propose an effective algorithm to generate test cases from grammars which cover a selected set of patterns extracted from the meta program under test. In this algorithm, we assume one constructor in a pattern corresponds to one labeled production rule of a context-free grammar. This initial algorithm guarantees full pattern coverage but not minimality of the set of test inputs. We shall see in the evaluation that the expected amount of tests generated is already so small that optimization in this direction is not necessarily an interesting avenue.

Given the set of pattern requirements corresponding to the meta program under test, the algorithm takes the following steps, for each pattern instance on the set:

1. Identify the open variables and their type (nonterminal);
2. Create a term for each identified variable for the given type using standard grammar-based test input generation;
3. Substitute the variable by the generated term in the pattern;
4. From the start symbol of the input grammar generate a sentence in the language including the previously generated term;
5. Add the result sentence to the test set.

On the first step, the algorithm identifies the open variables inside the term and generate terms to bind with them (step 2). Next step is apply a substitution of variable by the generated term (step 3). With this, we have a term that

Listing 1.2. Example of a Java sentence produced by the generation algorithm.

```
1 public class id0 {
2   static {
3     if (true)
4       return ;
5   }
6 }
```

matches the original patterns, but it is only part of a full input program. So, we generate a sentence using the input grammar (step 4). It is created from the start symbol and needs to reach the same nonterminal associated with the partial term that was generated on the last step. This step could be easily automated by adapting the traditional grammar-based testing algorithms. Finally, the resulting sentence is added to the test set (step 5).

To illustrate the algorithm execution, we use the implementation of CC on Listing 1.1 as example. The test requirement `\if(_,_)` is a pattern of type constructor and it has two open variables named as `_`. According to step 1 in the algorithm these two variable are detected and their nonterminal symbols are identified. The next step is to create a subterm for each of them: the first is a Java Boolean expression and the second a Java statement. For example these terms could be `true` and `return;`. On step 3 the variables are substituted by these terms and the result is a new subterm, in our example, `if (true) return ;`. The last step to produce the test data is to generate a sentence from the grammar start symbol including the previous subterm as an instance of its corresponding nonterminal. The Listing 1.2 shows a possible resulting Java sentence generated by this algorithm for the test requirement `\if(_,_)`.

In this paper we propose the use of patterns as reference for the definition of test data and a pattern-based coverage criterion which defines a set of test data requirements from a set of language patterns. These patterns may be extracted from different sources: from language or meta program specifications, from the patterns used in the code of the meta program, or even from a non pattern-based meta program from which some mining of patterns has been carried out.

The algorithm presented here was used in the evaluation process described in Section 4.

4 Evaluation

To evaluate our claim of effectiveness, we compare our new method of generating test inputs for meta programs to the state-of-art grammar-based testing methods in this section. Our evaluation is scoped to Rascal meta programs, which we assume to be representative for all meta programs which strongly depend on pattern matching.

The evaluations questions are:

Operator Code	Description
OP0	Remove pattern rewrite.
OP1	Remove pattern with action.
OP2	Remove `if` conditionals. Simple `if`.
OP3	Remove `if` conditionals. Remove `if` code block.
OP4	Remove `if` conditionals. Remove `else` code block.
OP5	Remove `while` conditionals.
OP6	Remove `for` conditionals.

Table 2. Table of Rascal Mutation Operators.

RQ1: *How efficient and effective are standard grammar-based testing techniques efficient for testing of Rascal meta programs?*
RQ2: *Is pattern-based testing more efficient and more effective than grammar-based testing for testing Rascal meta programs?*

4.1 Evaluation Method

We use two variables to measure the efficiency and effectiveness of this criterion: test set size for cover a given coverage criterion and mutation score, respectively.

The mutation score is a measure of effectiveness based on the concept of mutation testing [1,11]. Mutation testing provides a repeatable process for measuring the effectiveness of test cases and identifying disparities in (random) test sets. It involves the use of original software under test to create a variation of it that contains an artificial fault. Each fault inserted in original software produces a *mutant* and the specific fault injected is called *mutation operator*. After the application of a mutation operator the mutant is executed using the test set under evaluation, if the test indeed fails for one of the generated test cases the mutant is killed and the test set is good enough to detect that mutant operator. Otherwise, if all all tests set run without any failure, the mutant is alive and the random test case generation is deemed ineffective. For this last case there are two possibilities: the test set is indeed not good enough to trigger the mutant or the mutant is accidentally semantically equivalent to the original program. At the end, the percent of mutants killed by the test set is called *mutation score*.

The mutation process used in the evaluation is implemented in Rascal for Rascal. The set of mutation operators (Table 2) introduces bugs randomly, simulating programmers forgetting cases, making errors in patterns and making errors in the code that is triggered after a pattern is matched. Rascal is a pattern-oriented language, where patterns govern both data (binding) and control flow (conditionals) dependency. Patterns occur in the conditions of all structured control-flow statements and the parameters of functions (dynamic dispatch). Common Rascal programming errors are forgetting to update a pattern when a language has changed, accidentally overlapping patterns for which the code is then never or always executed, and writing overly restrictive patterns accidentally. The mutation operators are designed to highlight code which is executed

conditionally under a pattern, in order to make observable whether and how quickly the test set can trigger code which depends on (possibly buggy) patterns.

To evaluate the efficiency, we measure the number of test cases necessary to kill 100% of the mutants.

The comparison process. To provide an answer to the second question, we then run the same process using a test set that covers the pattern coverage criterion. The result of this second experiment is then compared with to the previous experiment. The comparison process is shown in Figure 1. The circles represent the process and the rectangles represent data. The process starts with the generation of a set of test data based on the language's context-free grammar using the algorithm in Section 3.2. An initial test data set is used with the software under test (Listing 1.1) to generate the expected results for this set of input. Since we use mutation testing to evaluate the test set effectivity we may use the not mutated version as the oracle. Given this generated test set and the generated oracle, the mutation process is started and each mutant is tested. This test process logs the two metrics (mutation score and number of test cases).

Selected languages and program under test. We use Java and Pico as the object languages for our experiment and a typical algorithm for computing the Cyclomatic Complexity (CC) [25] of a program for both languages (see Listing 1.1). CC is a basic but non-trivial algorithm, so this evaluation should be seen as an initial experiment and proof-of-concept. Lincke et al. [24] showed how even relatively basic and often used software metric implementations are inconsistent with each other, and thus broken, underlying the relevance of a feasible method for testing them thoroughly.

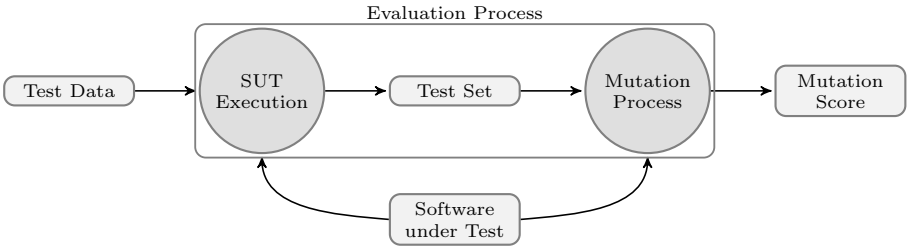

Fig. 1. An overview of evaluation process.

For the Java language, we used the *Java Specification Language* [17] and the ANTLR Java Grammar [7] specification as references. Since Java is a big language and therefore our initial demonstration and evaluation first focuses

[7] `https://github.com/antlr/examples-v3/blob/master/java/java/Java.g`

on a smaller language with a mini grammar: Pico [12]. The Pico language is a small educational programming language with a Rascal implementation[8]. This initial evaluation would also detect bugs in either our implementations or our evaluation methods. After this we continue with our evaluation on the Java language. Table 3 lists the sizes of the Pico and Java grammars we used.

Structure	**Size**	
	Pico	**Java**
Production Rules	18	272
Terminals	23	98
Nonterminals	9	142

Table 3. Grammar statistics for Java and Pico used in the evaluation.

To generate production coverage test data we used the LGen tool [27]. For both languages we generated the test data and after that ran the software under test, the cyclomatic complexity algorithms, to create the expected result and produce the complete test set.

4.2 Results for Pico Cyclomatic Complexity

For the Pico language the results obtained by the evaluation method is presented in Table 4. Using the production coverage test set we reach 100% of mutation score. It means that all mutations generated by mutation process have been killed by the test set. The pattern coverage test set we also killed all mutants, but in this case with a lower number of test cases. This results provides an initial evaluation about the pattern coverage's efficiency but without enough confidence on it because its small scale and low complexity of the Pico language.

Coverage Criterion	**Test Set Size**	**Mutation Score**
Production Coverage	10	100%
Pattern Coverage	2	100%

Table 4. Results obtained by the evaluation using the Pico language.

4.3 Results for Java Cyclomatic Complexity

The results obtained from Java case are shown in Table 5. In this case, we observe a reduction over on the number of test cases using the pattern coverage criterion. This is related to the amount of patterns used on program 1.1. Furthermore, the mutation score for this coverage criterion shows an increase in relation to the production coverage criterion.

[8] `http://tutor.rascal-mpl.org/Recipes/Languages/Pico/Pico.html`

Coverage Criterion	Test Set Size	Mutation Score
Production Coverage	85	8.33%
Pattern Coverage	12	83.33%

Table 5. Results obtained by the evaluation using the Java language.

4.4 Discussion and Threats to Validity

Even given the small size and complexity, from the Pico experiment we learn that pattern coverage may reduce the amount of required test cases dramatically without sacrificing test quality. This small demonstration can not be generalized to other languages or meta programs, but it motivates to continue investigating and served as an integration test for our experimental setup.

The Java language results show that the approach will scale to full programming languages. We observe that indeed the number of test cases drops significantly without loss of test quality.

The main threat to validity of the above observations is the size and complexity of the meta program under test, which is not representative of the broader set of meta programs we wish to target. These initial results are promising nevertheless. We expect future investigation on larger meta programs to produce more specific patterns (which does not incur an overhead since pattern extraction is linear in the size of the meta program). More specific patterns will likely lead to larger but not more test cases. We hypothesize that pattern-based testing will still outperform exhaustive grammar-based testing in the number of test cases, while we need to find out experimentally what the mutation scores will show. The mutation score is influenced by the internal complexity of the meta program which may use other kinds of predicates and queries next to pattern matching to guide control flow. More complex meta programs will also require us to extend the set of mutation operators to generate representative mutations. Our future target is a group of type analysis and type inference tools for Rascal, Java and PHP which we wish to test exhaustively.

We believe that future work to extend pattern-based testing to other application domains based on schemata and patterns may be fruitful (XML processing, DOM manipulation and model driven engineering)

5 Related Work

Grammar-based testing has been used in compiler testing for many years [5, 20]. Compilers are a specific kind of meta programs and could be tested using similar techniques. The YouGen tool [14] generates test data based on grammar definitions, controlling depth of derivation trees based on annotations. The major difference is about the algorithm based on enumeration and filtering of the derivation trees. This also reduces the number of test data generated, but the filtering is guided only using information from the grammar. Our approach also uses information from the program under test.

In the same direction, the XTextGen [13] generates test-data based from grammars. This tool has a different approach to generate the data. It uses Cantor pairing and mandatory multiplicity control. The generation process is split into two phases. First, an enumeration process and then a semantics-directed post-processing over the result a set of transformation. With XTextGen it should be possible, in principle, to simulate pattern-based testing as we propose in the current paper. An appropriate encoding would have to be developed to do so, which would amount to an alternative implementation of our sentence generation algorithm.

Jagannath et al. [16] propose several ways of reducing the cost of bounded exhaustive testing (the generalization of grammar-based testing) by test prioritization and by merging smaller inputs into fewer bigger inputs. Our approach also reduces the cost of exhaustive testing, by selecting the right test inputs to run, but in a completely different way.

SafeRefactor [10] also uses bounded exhaustive search for generating test programs (in AST format and for the purpose of refactoring tools). A key ingredient is programmable specialization of the generated ASTs to better fit the specific meta program under test. This framework, which was elaborated on later for scalability [26, 30], could be extended with our approach to automatically feedback patterns back into the AST generators.

With concolic testing [29] a similar effect of selecting the right test cases could be achieved as our approach can. Concolic testing is based on symbolically simulating a program for a given test input and using a SMT solver or theorem prover to generate the next input which will cover a different execution path than the previous test input did. One would need to add a theory for grammar and pattern matching to the solver and a simulation engine for the meta language for this approach to work. Our approach is different and much more lightweight, neither requiring a hard-to-obtain efficient solver nor a symbolic execution engine, just a pattern extraction tool and a grammar-based sentence generator.

6 Conclusions and Future Work

In this paper, we presented a new coverage criterion for test case design, pattern coverage) and its preliminary evaluation of effectiveness in the context of pattern-based meta programs. The evaluation considered the amount of test cases and mutation score, taking as reference traditional grammar-based testing. We conclude that our experiments indicate a significant reduction in necessary test cases to achieve coverage, while improving the quality of the tests in terms of mutation score. Further evaluation with a richer set of meta programs and mutation operators is planned in the next steps of the research.

This work is part of a more general research direction in which we are investigating lightweight techniques to more effectively test meta programs: extracting constraints from the source code of the programs under test (or their specifications) to direct a sentence generator.

7 Acknowledgments

This work is partly supported by CNPq grants 237049/2013-9 and 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES, `www.ines.org.br`).

References

1. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.
2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.
3. E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In F. Baader, editor, *Term Rewriting and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer Berlin Heidelberg, 2007.
4. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. *Electronic Notes in Theoretical Computer Science*, 4(0):35 – 50, 1996.
5. A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information Software Technology*, 39(9):617–625, 1997.
6. M. Brand, A. Deursen, J. Heering, H. Jong, M. Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.
7. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52 – 70, 2008. Special Issue on Second issue of experimental software and toolkits (EST).
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.
9. K. Czarnecki and U. W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000.
10. B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. ACM.
11. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
12. A. V. Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach: Vol. V*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.
13. J. Härtel, L. Härtel, and R. Lämmel. Test-data generation for xtext. In *Software Language Engineering*, pages 342–351. Springer, 2014.

14. D. Hoffman, D. Ly-Gagnon, P. Strooper, and H.-Y. Wang. Grammar-based test generation with yougen. *Software: Practice and Experience*, 41:427–447, 2011.
15. P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.
16. V. Jagannath, Y. Y. Lee, B. Daniel, and D. Marinov. Reducing the costs of bounded-exhaustive testing. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 171–185, Berlin, Heidelberg, 2009. Springer-Verlag.
17. G. L. S. J. G. B. James Gosling, Bill Joy. *The Java Language Specification*, volume 1. Addison Wesley, 3rd edition, 2005.
18. P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:168–177, 2009.
19. P. Klint, T. van der Storm, and J. Vinju. Easy meta-programming with rascal. In J. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer Berlin / Heidelberg, 2011.
20. A. S. Kossatchev and M. A. Posypkin. Survey of compiler testing methods. *Program. Comput. Softw.*, 31(1):10–19, 2005.
21. R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In M. Ü. Uyar, A. Y. Duale, and M. A. Fecko, editors, *Testing of Communicating Systems*, volume 3964 of *Lecture Notes in Computer Science*, pages 19–38. Springer Berlin Heidelberg, 2006.
22. D. Landman, A. Serebrenik, and J. Vinju. Empirical analysis of the relationship between CC and SLOC in a large corpus of Javah methods. In *30th IEEE International Conference on Software Maintenance and Evolution, ICSME 2014*, 2014.
23. X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, Dec. 2009.
24. R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 131–142, New York, NY, USA, 2008. ACM.
25. T. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.
26. M. Mongiovi, G. Mendes, R. Gheyi, G. Soares, and M. Ribeiro. Scaling testing of refactoring engines. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 371–380, 2014.
27. A. M. Moreira, C. Hentz, and V. de Menezes Ramalho. Application of a syntax-based testing method and tool to software product lines. In *7th Brazilian Workshop on Systematic and Automated Software Testing, SAST 2013*, 2013.
28. M. Odersky, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, and et al. An overview of the scala programming language. Technical report, École Polytechnique Fédérale de Lausanne, 2004.
29. K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.
30. G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *IEEE Trans. Software Eng.*, 39(2):147–162, 2013.